ParaPLL: Fast Parallel Shortest-path Distance Query on Large-scale Weighted Graphs

---

Final Project for the course of CS309 – 2021

Goals: To study, analyse, and implement the algorithm

Abhinav Reddy (190001007)
Kola Sai Ram (190001026)
Kushaan Gowda (190001031)
Mayank Raj (190001035)

1. Introduction

Shortest-path distance is one of the most fundamental metrics for determining relationships between vertices in a graph. For example, a distance query can find the best path between two nodes in a network. In a social network, the distance between two individuals might signify their closeness or similarity. In web graphs, this distance can suggest the context-based similarity and recommend similar web pages.

The early techniques for querying distance relied on a breadth-first search (BFS) algorithm (for unweighted graphs) or Dijkstra's algorithm (for weighted graphs). However, due to their time complexity of O(n2), these algorithms may have scalability problems for large graphs.

As a result, the authors presented ParaPLL, a fast, precise shortest-path distance query framework for large-scale weighted graphs. The performance issues in parallel distance-indexing of the shortest path are addressed by ParaPLL, which is based on parallelizing PLL. ParaPLL can take full advantage of the processing power provided by several levels of parallelism, such as multi-core CPUs.

2. Problem Statement

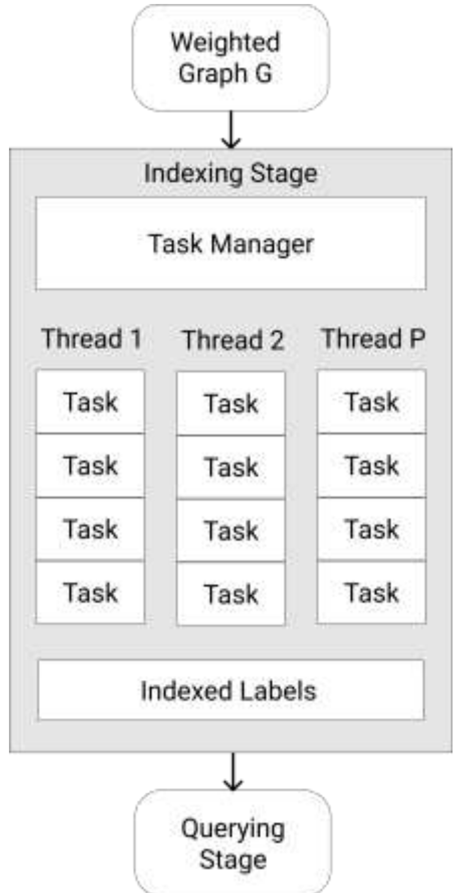
Given an undirected graph G, we must construct an index that efficiently answers distance queries. We also need to ensure that the design scales well in terms of multi-threading.

3. Notations

Symbol	Description
G	An undirected graph
L[u][v]	Shortest distance between source vertex v and destination vertex u
weight(u, v)	Weight of edge connecting vertex u and v

4. ParaPLL

ParaPLL is a two-stage algorithm that consists of an indexing stage and a querying stage, as shown in the figure.



4.1 Indexing Stage

To obtain the indexes (labels), we employ Pruned Dijkstra's algorithm. Label L[u][v] is defined as the shortest distance with u as the destination and v as the source vertex. We compute labels L[u][vk] given an undirected graph G and a vertex vk. Except for line 6b, where we include pruning, most of the steps in the pseudocode are the same as those in Dijkstra's algorithm. To get the shortest distance, we use the Query function (described in the following section). If this distance is smaller than D[u], it indicates that our path is already shorter than the one produced by computing D[u], and we thus skip this vertex.

Algorithm 1: Pruned_Dijkstra(G, v_k , L)
Input: graph G, vertex $v_k \in V$, labels L
Output: updated labels L

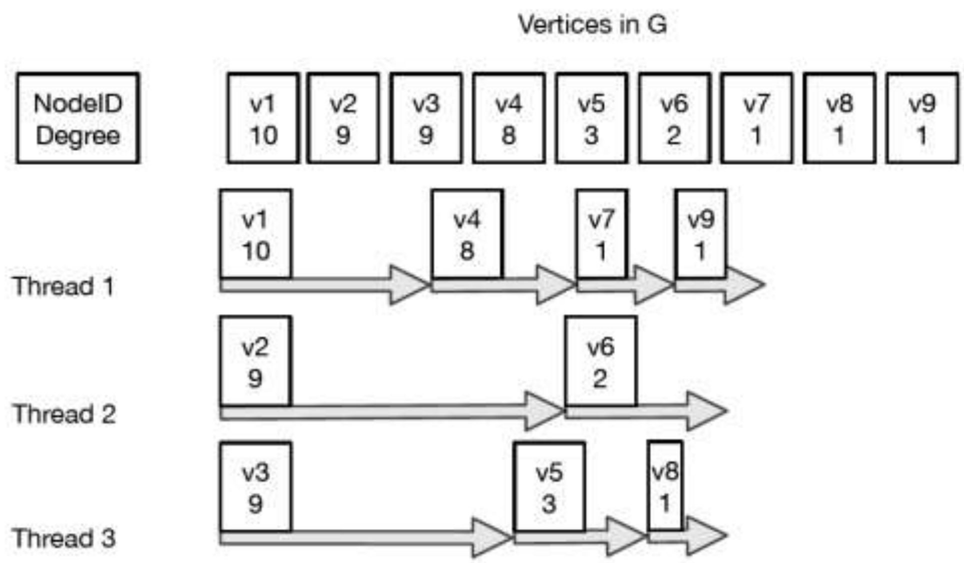
1. D is an array such that D[i] represents the shortest distance between vertex i and v_k
2. D[v_k] ← 0
3. D[v] ← ∞ for v ∈ V \{v_k}
4. Q is a priority queue which stores vertices i based on D[i] in ascending order
5. Enqueue v_k to Q
6. While Q is not empty do
    a. Dequeue u from Q
    b. if Query(v_k , u, L) ≤ D[u] then
        i. continue
    c. L[u][v_k] ← D[u]
    d. for neighbour w of u such that D[w] < D[u] + weight(w,u) do
        i. D[w] ← D[u] + weight(w,u)
        ii. Enqueue w to Q

4.2 Querying Stage

Given the indexes (labels) computed in the indexing stage, we can answer an incoming distance query with Query(s, t, L), which is defined as follows:
$\min\{L[u][s] + L[u][t] \mid u \in V\}$.
This statement means that we are trying to find the shortest distance between vertex s and vertex t via vertex u. So, if we calculate this for all vertices $u \in V$, we can find the shortest-path distance between s and t.

4.3 Parallelizing

In the main function, we introduce thread-level parallelism. Given that we can execute our algorithm on p threads, we dynamically allocate a vertex to each thread and perform pruned Dijkstra's algorithm on that vertex. The same is depicted in the figure below.



Because the results were slightly better in this case, we sort the vertices in descending order according to their outdegree.

Algorithm 2: ParaPLL_Main
Input: Graph G, number of threads p
Output: L

1. $L[u][v] \leftarrow \infty$ for $u,v \in V$
2. $Q \leftarrow$ a queue with n ordered vertices

3. While Q is not empty do
    a. for k = 1, 2, ..., p in parallel do
        i. Dequeue v from Q
        ii. Pruned_Dijkstra(G, v , L)
4. Return L

5. Evaluation

5.1 Dataset

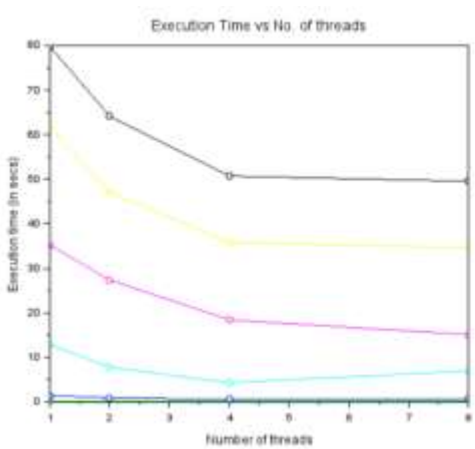
Due to lack of hardware support, we have restricted ourselves to use the datasets mentioned below, containing a weighted undirected graph with the following configuration:

- D1 (20 vertices, 98 edges)
- D2 (889 vertices, 2914 edges) (Subset of wiki-vote dataset)
- D3 (1000 vertices, 2730 edges)
- D4 (1500 vertices, 13440 edges)
- D5 (2000 vertices, 23052 edges)
- D6 (2500 vertices, 26824 edges)
- D7 (3000 vertices, 21968 edges)

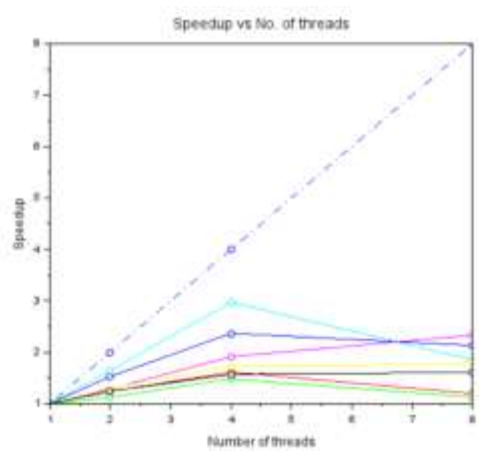
5.2 Results

ParaPLL Algorithm

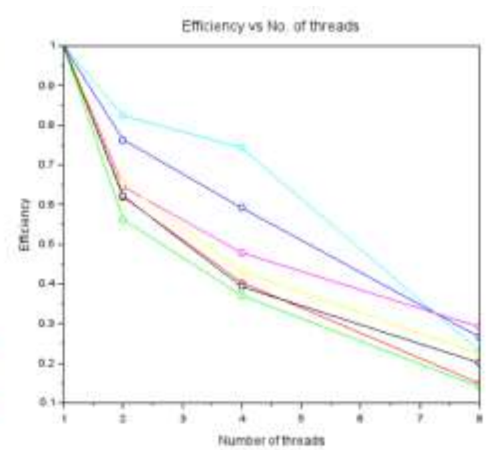
Execution Time



Speedup



Efficiency



Legend

Color	Dataset	Color	Dataset
Red	D1	Magenta	D5
Green	D2	Yellow	D6
Blue	D3	Black	D7
Cyan	D4	Dashed Line	Ideal Case

Results Table

D1			
Number of threads	**Execution time (in secs)**	**Speedup**	**Efficiency**
1	0.000799937	1	1
2	0.000645766	1.238741278	0.6193706389
4	0.000495287	1.615097913	0.4037744782
8	**0.000667954**	1.197592948	0.1496991185

D2			
Number of threads	**Execution time (in secs)**	**Speedup**	**Efficiency**
1	0.235291	1	1
2	0.209113	1.125185904	0.5625929521
4	0.158591	1.483634002	0.3709085005
8	**0.208363**	1.129235997	0.1411544996

D3			
Number of threads	**Execution time (in secs)**	**Speedup**	**Efficiency**
1	1.49081	1	1
2	0.975834	1.527729101	0.7638645507
4	0.629533	2.368120496	0.5920301239
8	**0.69846**	2.134424305	0.2668030381

D4			
Number of threads	**Execution time (in secs)**	**Speedup**	**Efficiency**
1	12.9514	1	1
2	7.85262	1.649309402	0.8246547012
4	4.35646	2.972918379	0.7432295947

8	**6.98764**	1.853472703	0.2316840879
D5			
Number of threads	**Execution time (in secs)**	**Speedup**	**Efficiency**
1	35.3318	1	1
2	27.3813	1.290362401	0.6451812003
4	18.4591	1.914058649	0.4785146621
8	15.133	2.334751867	0.2918439833
D6			
Number of threads	**Execution time (in secs)**	**Speedup**	**Efficiency**
1	61.7828	1	1
2	47.1364	1.310723772	0.6553618859
4	35.7569	1.727856721	0.4319641803
8	34.6514	1.782981351	0.2228726689
D7			
Number of threads	**Execution time (in secs)**	**Speedup**	**Efficiency**
1	79.8431	1	1
2	64.1883	1.243888684	0.6219443419
4	50.7282	1.57393915	0.3934847876
8	49.5531	1.611263473	0.2014079341

5.3 Inference

As we increase the number of threads, the execution time decreases. As shown in the graphs above, the speedup isn't as high as it should be, and efficiency drops as the number of threads increases.

The execution time for eight threads is larger than that for four threads, as shown in the results table for datasets D1, D2, D3, and D4. These oscillations (marked in bold) represent the overhead introduced by thread management algorithms, decreasing the efficiency. Another reason could be that our hardware is dual-core two-threaded, so exceeding a certain number of threads lowers the efficiency. Not all of the code in the suggested technique is parallelized, restricting efficiency as the number of threads increases.

6. Conclusion

In this paper ParaPLL has been used to address the scalability barrier in the shortest-path distance query problem. A multi-threading framework is used to speed up the indexing part of the shortest-path distance query. This method can theoretically achieve a linear speedup. This theoretical bound, however, could not be attained due to the overhead incurred by multi-threading. This study can be extended by testing this technique on larger datasets with appropriate hardware support and looking into ways to parallelize pruned dijkstra's algorithm.